Reliable Solution of Special Event Location Problems for ODEs

L. F. Shampine[1,2], I. Gladwell[2,3]
Mathematics Department
Southern Methodist University
Dallas, Texas 75275

R. W. Brankin
Numerical Algorithms Group Ltd.
Mayfield House
256 Banbury Road
Oxford OX2 7DE

February 13, 1997

[1] Partially supported by the Applied Mathematical Sciences program of the Office of Energy Research under DOE grant DE–FG05–86ER25024
[2] Partially supported by NATO Research Grant 898/83
[3] Partially supported by a Royal Society/SERC Industrial Fellowship and by the Numerical Algorithms Group Ltd.

Abstract

Computing the solution of the initial value problem in ordinary differential equations (ODEs) may be only part of a larger task. One such task is finding where an algebraic function of the solution (an event function) has a root (an event occurs). This is a task which is difficult both in theory and in software practice. For certain useful kinds of event functions it is possible to avoid two fundamental difficulties. It is described how to achieve the reliable solution of such problems in a way that allows the capability to be grafted onto popular codes for the initial value problem.

Categories and Subject Descriptors: G.4 [Mathematical Software]: Reliability and robustness, portability; G.1.7 [Numerical Analysis]: Ordinary differential equations — initial value problems. Root-finding.

General Terms: Algorithms

Additional Key Words and Phrases: Dense output, event location.

1 Introduction

Computing the solution, $y \epsilon R^n$, of the initial value problem in ordinary differential equations,

$$y' = f(x, y), \ a \leq x \leq b, \ y(a) = y_a \tag{1.1}$$

may be only part of a larger task. Possibly the most common such task is to find either a first point $t_0 > a$ or a set of points $\{t_s\}$, $a < t_0 \leq t_1 \leq \ldots$, such that one of the equations

$$g_j(x, y(x), y'(x)) = 0 \qquad j = 1, 2, \ldots, m \tag{1.2}$$

is satisfied at $x = t_s$. The g_j are called "event functions", and event j is said to occur at t_s when t_s is a root of the jth event function. This paper is concerned with a class of such problems which have each event function g_j defined as a polynomial in either y or y'. Specifically, we assume that g_j in (1.2) has the form either

$$g_j(x, y, y') \equiv y_{k_j}(x) - \alpha_j \tag{1.3}$$

or

$$g_j(x, y, y') \equiv y'_{k_j}(x), \tag{1.4}$$

where $y_{k_j}(x)$ is a component of the solution vector $y(x)$. More general polynomial forms of g_j can be treated by the same techniques at the expense of rather more complicated machinery. Common problems such as

(i) finding where a component of the solution assumes a given value, and

(ii) finding where a component of the solution has an extremum

have the form (1.3) or (1.4) with a single event function. We allow several event functions of both forms at the same time, and so can solve more complicated problems such as

(iii) tabulating values of a dependent variable y_j;

(iv) determining the location of switching points or points of discontinuity defined in terms of linear functions of a dependent variable y_j; and

(v) determining zeros of a general event function $q(x, y) = 0$ by adjoining a differential equation for q to the system (1.1).

The difficulty of the event location problem is often not appreciated. In the next section we investigate the theoretical difficulties. Two fundamental difficulties are that, with the usual approaches, one cannot be sure of noticing an event at all, and that if one realizes that an event has occurred, one cannot be sure of finding the first occurrence. For the special problems we consider

here, we present an efficient way to overcome both these difficulties, at least in principle; other difficulties remain.

In Section 3 we state the key observation and describe how to exploit it. The presence of more than one event function causes many complications. We present a software design which is reasonably easy to use and which provides considerable flexibility. We have produced codes for event location that can be used with many of the popular codes for the initial value problem. Our goal was to add the capability of locating events without altering the integrator itself. In Section 4 we describe how we accomplish this and in Section 5 we discuss the codes themselves and an important application of them.

2 The Problem and its Difficulties

Popular codes for the initial value problem (1.1) step from a to b, producing approximations y_i, y_i' to $y(x_i), y'(x_i)$ at a set of points $a = x_0 < x_1 < \ldots$. It is usual with such codes to test the values of each event function of (1.2) for different signs at x_i and x_{i+1}. A change of sign in any one indicates that an event has occurred in $[x_i, x_{i+1}]$. Adams and backward differentiation formulas (BDF) codes produce a polynomial $p(x)$ that approximates $y(x)$ on the whole interval $[x_i, x_{i+1}]$. It is natural then to compute the first root of

$$g_j(x, p(x), p'(x)) = 0$$

as an approximation to the location of event j. The popular Runge-Kutta formulas produce solutions only at the mesh points x_i. This approach to the event location problem has been an important reason for the recent work aimed at providing these formulas with polynomial approximate solutions valid for all of $[x_i, x_{i+1}]$.

The way of locating events just described is so natural that many have been led to think that the task itself is easy. This is far from true. The mesh points x_i are chosen to provide efficiently approximations to $y(x)$ and $y'(x)$ of a specified accuracy. Because the event functions (1.2) do not influence the selection of the mesh, the spacing may not be at all appropriate for locating the positions of the events, t_s, that is, the roots of (1.2). If an even number of roots, counting multiplicity, should occur between x_i and x_{i+1}, they will not be noticed. Should the presence of an event be noticed, there is in general no way to be certain the root-solver will find the *first* root. Of course, one might monitor the behavior of the g_j more closely to reduce the chance of missing a root. A particularly careful algorithm is that of Watts [12, 13]. It can be expensive, and is not guaranteed to succeed, but it is unlikely to be deceived by any but near-pathological cases.

An event function of the form $q(x, y(x))$ can be converted to the form we consider when analytical partial derivatives are available. To do this, a new dependent variable $z(x) = q(x, y(x))$ is added to the system (1.1) by noting

that

$$\begin{aligned} z(a) &= q(a, y_a) \\ \frac{dz}{dx} &= q_x(x, y(x)) + q_y(x, y(x))f(x, y(x)). \end{aligned}$$

We then look for the event $z(x) = 0$. By this conversion we force the integrator to select step sizes appropriate to $q(x, y(x))$, and as a consequence, the location of events is then much more reliable. Unfortunately, it may be expensive to compute $q(x, y(x))$ accurately in this way. This cost would have to be paid even if the event does not occur in $[a, b]$. Also, the partial derivatives q_x and q_y may be difficult to obtain.

As we have stated, in the event location problem equation (1.1) is to be integrated at least until the first event occurs. A common use of event functions is to determine when the function $f(x, y)$ is to be redefined. For example, we might have one set of differential equations which describe the temperature in a room until the temperature drops so low that a thermostat switches on a heater. The new situation is described by a new set of equations. In this example, the definition of f clearly depends on the history of the integration; it is natural to locate the event, and then to restart the integration with the new definition of f. A more deceptive problem occurs when f may be redefined, but the various possible definitions do not depend on the history of the integration. For example, suppose that $f(x, y) = y$ for $|y| \leq 1$, and $f(x, y) = sign(y)$ otherwise. It is a temptation to program f in a straightforward fashion, but in large measure this defeats the efficient location of events. The difficulty is that codes for the initial value problem expect f to be smooth. Programming this f so that its value is determined from a simple test on the magnitude of y presents the integrator with a function f that is continuous but lacks a continuous derivative. If the integrator steps from a value of $y < 1$ to a value $y > 1$, the function f is not smooth enough that the basic theory underlying the construction of the integrator applies. What happens then is unpredictable. The integrator may not "realize" it has produced an inaccurate solution. Mathematical software for solving differential equations tries to recognize and cope with problems for which it is not designed. Quality codes are likely to recognize an inaccurate solution, reject the step, and try again with a smaller step. It is often the case that such codes will repeatedly try steps that cross the event and, on their rejection, steps that fall short but succeed. In this way, eventually the integrator steps past the event with a step size so small that the result is accurate. Proper use of a code with event location ability avoids this clumsy and unreliable handling of the discontinuity. For an initial value $|y| < 1$, the smooth function $f(x, y) = y$ should be integrated until one of the events $y = 1$ or $y = -1$ occurs. After the location of this event has been determined, f should be redefined and the intgration restarted with a new f that is smooth, namely $f(x, y) = \pm 1$, as the case may be. This way of proceeding is not only more efficient, it furnishes a reliable solution of the task for which it is intended. In this paper, then, we

suppose that the function f is smooth and that if f is redefined because of an event at $x = \tau$, a new integration will be initiated from this point.

How accurately we can locate an event depends on how accurately we integrate the underlying differential equations and on the conditiong of the root. The difficulties are present even with the simple event functions we treat, so we discuss the matter in the specific context of finding a root τ of the event function $y_j(x) - \alpha$. At each step the code produces a polynomial $p(x)$ approximating the solution component $y_j(x)$. The event τ is then approximated by a root $\bar{\tau}$ of the equation $p(x) - \alpha = 0$.

A minor source of error in the location of an event is in the computation of $\bar{\tau}$. Some codes require the user to specify how accurately $\bar{\tau}$ is to be computed. We, and other authors, prefer to compute $\bar{\tau}$ as accurately as possible in the precision available. It is then reasonable to suppose that the computed $\bar{\tau}$ satisfies $p(\bar{\tau}) - \alpha = 0$ exactly. Though not quite true, the discrepancy is far smaller than the other errors we consider.

We relate $y_j(\tau)$ to its approximation $p(\tau)$ by

$$0 = y_j(\tau) - \alpha = (y_j(\tau) - p(\tau)) + (p(\tau) - \alpha).$$

If $\bar{\tau}$ is a simple root, we can approximate the second term here by

$$p(\tau) - \alpha \approx p(\bar{\tau}) - \alpha + (\tau - \bar{\tau})p'(\bar{\tau}) = (\tau - \bar{\tau})p'(\bar{\tau})$$

which leads immediately to an approximation of the error in τ,

$$\tau - \bar{\tau} \approx -(y_j(\tau) - p(\tau))/p'(\bar{\tau}). \tag{2.1}$$

If $p'(\bar{\tau})$ is "small", the root τ is poorly determined. It is easy to extend the argument to deal with multiple roots, and so come to the conclusion that all multiple roots are ill–conditioned. What differs here from the usual algebraic case is the term $y_j(\tau) - p(\tau)$, which represents the true, or global, error of the integration of (1.1). Codes for the initial value problem do not even try to control this error directly. It is, at best, of a size comparable to the local error tolerance used in the integration. Generally we cannot expect to locate an event any more accurately than the integration tolerance, and if the problem (1.1) is ill–conditioned, we may not achieve even close to this accuracy.

It would be very helpful to the user of a code with an event location capability to have an estimate of the error in the reported $\bar{\tau}$. In our code we provide an estimate of the quantity multiplying the global error $y_j(\tau) - p(\tau)$ in (2.1) and the equivalent quantity for multiple roots. The esimate (2.1) refers to event functions of the form (1.3). It is easy to obtain analagous results for event functions of the form (1.4), and our code estimates the condition of these problems in a similar way. To estimate the error in the location of τ, we would need an estimate of the global error in $y_j(\tau) - p(\tau)$ in the one case, and of $y_j'(\tau) - p'(\tau)$ in the other. We are not aware of a satisfactory way to estimate these

quantities. It is commonly assumed that these global errors are comparable to the local error tolerance. This assumption, along with the condition numbers we provide, yields a crude, but useful, indication of the error in $\bar{\tau}$.

Some of the implications of these facts are discomfiting. It is perfectly possible that an event occurs when the integration is done at one tolerance and does not occur at a different tolerance. This is unpleasant, but what is worse is that this can happen at the *same* tolerance if the integration is performed differently, as for example if different output requirements alter the choice of step size. In particular, repeating an integration with the code stepping to the alleged location of an event will ordinarily result in a value of $g(x, y(x), y'(x))$ different from that obtained using the polynomial approximation $p(x)$ between the mesh points. A related matter is that in general

$$p'(x) \neq f(x, p(x)),$$

so that at the computed event position $\bar{\tau}$

$$g(\bar{\tau}, p(\bar{\tau}), f(\bar{\tau}, p(\bar{\tau}))) \neq 0,$$

contrary to what one might expect.

It is not often discussed why typical codes allow only $y(x)$ and $y'(x)$ in (1.2). A large part of the answer is that they suffice for many of the problems seen in practice. However, there is a fundamental difficulty with higher order derivatives. A polynomial may fit $y(x)$ well, yet have derivatives that do not exhibit even qualitative agreement with those of y. The Adams methods are based on a polynomial approximation to $y'(x)$, so it is perfectly natural to include it in (1.2). With other methods, the presence of $y'(x)$ in (1.2) is more debatable. An expensive way to handle the first derivative is to eliminate it by solving instead

$$g(x, y(x), f(x, y(x))) = 0.$$

This is reliable for non-stiff problems, but should be avoided for stiff problems. As is well-known, the $p'(x)$ associated with the BDF codes is ordinarily a far better approximation to $y'(x)$ than is $f(x, p(x))$. Any derivative can be handled if analytical partial derivatives are available by adding the derivative as a new variable to the system (1.1) rather like the event function q was added as a variable above. Proceeding in this way causes the codes to produce an *independent* polynomial approximation to the derivative to the specified local accuracy. Unfortunately, obtaining the equation satisfied by the derivative may be difficult.

Besides all these fundamental difficulties there is another that arises from the polynomial approximations produced by popular codes. The polynomial produced for the popular Fehlberg (4,5) Runge-Kutta formulas by Horn [8] and that in the Adams code of Shampine and Gordon [10] do not connect at the mesh points x_i to form a globally continuous function. Those of the BDF and

Adams code of Gear [5] form a continuous function, but the first derivative has jump discontinuities. The jumps seen in these codes are comparable in size to the local error tolerances. Obviously, the scheme described for locating events can exhibit anomalous behavior when such approximate solutions are used. To remedy this difficulty, a number of authors have recently supplied algorithms with C^1 piecewise polynomial approximations for Runge-Kutta [4, 11], Adams [14], and BDF [1, 13] codes.

3 A Root-Finding Algorithm

In this section we describe the technique we use to solve special event location problems. Where appropriate or necessary we describe some details of the code, but it is our aim here only to present the important ideas and a suitable structure for their realization. The code itself is discussed in more detail in Section 5 and presented in [2].

Consider the task of integrating (1.1) until the first position satisfying (1.2) is located. The key observation is that for event functions of the form (1.3) and (1.4), when $y(x)$ is approximated by a polynomial $p(x)$, the function $g_j(x, p(x), p'(x))$ is itself a polynomial. Of course, this is true of other kinds of event functions as well, but we restrict ourselves to the forms (1.3) and (1.4) because they allow us to solve interesting practical problems and still provide a (relatively) simple user interface. Our approach also generalizes to other kinds of approximations. We have found [6, §4] that for some purposes, a piecewise rational approximation to $y(x)$ offers advantages. It is easy to see that for event functions of the form (1.3) and (1.4), rational approximation also leads to finding roots of polynomials. We shall say no more about these obvious generalizations. Using Sturm sequences, say, we can, in principle, answer the question, "Is there a root of the polynomial $g_j(x, p(x), p'(x))$ in the interval $(x_i, x_{i+1}]$?" It is also possible to be sure of computing the *first* root if one is present by using, for example, bisection and Sturm sequences to test for the presence of a root. In what follows we describe one way to exploit this observation.

First we need to be clear about what we mean by the " first position" at which equation (1.2) is satisfied. It is quite possible, and indeed fairly common in practice, that (1.2) is satisfied at the *initial* point $a = x_0$. Of course, the user is able to check this without our assistance, and so there is no need to report it. For the integration of (1.1) we intend only to employ codes that can be used in a mode such that on each call to the routine, the integrator steps from the current point x_i to an internally chosen point x_{i+1} in the direction of integration. We intend to search the current integration interval for the next occurence of an event defined by (1.2). Now if we define the current integration interval as the *half open* interval $(x_i, x_{i+1}]$, then overall we lose no points from the range of integration, except possibly the initial point $x_0 = a$. This definition has two significant advantages. First, our technique for determining the position of

events is based on a Sturm sequence algorithm [9] and the count of the number of zeros given by this algorithm is always defined on such an interval. Second, if we locate an event at a point $\bar{t}\varepsilon(x_i, x_{i+1}]$ and we wish to go on to locate the next event in the direction of integration, then the interval to be searched initially is naturally $(\bar{t}, x_{i+1}]$. In our codes we provide two basic modules:

(i) One module is designed solely for checking whether there are any events (1.2) in $(x_i, x_{i+1}]$. This module takes as input the interval $(x_i, x_{i+1}]$ and a shifted power series representation of all the event functions g_j. It uses a standard Sturm sequence algorithm to determine precisely how many occurrences, counting multiplicities, there are of each event in $(x_i, x_{i+1}]$.

(ii) A second module is used for locating the first event in a user specified interval $(c, d] \subset (x_i, x_{i+1}]$ after a call to the module described above has determined that one or more events do occur in $(x_i, x_{i+1}]$. The module's output is an accurate estimate of the first position $\bar{t}$ of *any* event in $(c, d]$, its multiplicity and/or type, depending on context, a condition number and an interval $(\bar{c}, \bar{d}] \subset (c, d]$ such that $\bar{t}\varepsilon(\bar{c}, \bar{d}]$ and such that in the search for $\bar{t}, (\bar{c}, \bar{d}]$ is the largest interval seen which contains just one event.

When the solution $y(x)$ is approximated by $p(x)$, there will be some change in the position of the events and possibly in their multiplicity, especially if the original problem has events which are of high multiplicity or are not well separated. We assume here that we are solving

$$g_j(x, p(x), p'(x)) = 0, \qquad j = 1, 2, \ldots, m.$$

As we stated in Section 2, we return condition numbers for the roots that can be used to gain some impression of their accuracy.

Because we want our algorithm for locating the first position of any event to be both efficient and robust, we have tried to devise an algorithm that converges rapidly when we are satisfied that we have reduced the task to locating an event for a single event function, and is cautious in other less frequently occurring circumstances. If we did not insist on this efficiency, we might employ a simpler algorithm. For example, we could apply a standard code for computing the roots of polynomials to locate all occurrences of all events and then rank the results to determine the location of the first event. This approach would be palpably inefficient even if we were first to check which events occur in $(x_i, x_{i+1}]$, or as we shall say, which event functions are "active". Important to efficiency is the fact that in many applications there will be just one event function, that is, $m = 1$ in (1.2). Also, whether or not $m = 1$, we expect that in many integration intervals, there will be no events at all; that when an event does occur, it will usually be the only one in the interval; and that if several events occur in one integration interval, they will usually be isolated, relative to machine roundoff.

In our codes, whenever we have isolated a single active event function with just a single occurrence (and hence computed $(\bar{c}, \bar{d}]$), we use a standard bisection/secant algorithm [3], rewritten in reverse communication form, to locate

the position $\bar{t}$ of the event accurately. To get to this stage, we first identify all the events active in the interval $(c, d]$ of current interest. If there are any, we make a list of them:

$$\{g_{k_r}\}_{r=1}^{\bar{m}},\ \bar{m} \leq m,\ 1 \leq k_1 < k_2 < \cdots < k_{\bar{m}} \leq m.$$

Next we use a bisection algorithm to identify a point $\underline{d}\varepsilon(c, d]$ such that, for the first element of the active events list, the event occurs only once in $(c, \underline{d}]$.

Suppose we have a current interval $(\hat{c}, \hat{d}]$, (initially set to $(c, \underline{d}]$). We may have only a single active event, in which case we set $(\bar{c}, \bar{d}] \equiv (\hat{c}, \hat{d}]$ and we use the bisection/secant algorithm to locate $\bar{t}$ precisely. Even if we have more than one active event it is worth proceeding in a similar way. There are a variety of possibilities. These include using the bisection/secant algorithm on each of the active events in turn, followed by ranking the computed positions to find the first. A second possibility is to locate the position $\hat{t}$ of the first of the active events accurately, and then to check if this position is the first over all the events; if it is not, we then find the active events on $(\hat{c}, \hat{t}]$ and start again. We have, however, adopted an approach that is generally more efficient. We choose arbitrarily to work with the first of the active events and to proceed with one iteration at a time of the bisection/secant algorithm for this event. We compute an estimate, e, of the position of this event by a call to the bisection/secant code. There are then several possibilities. If the estimate is to the right of the locations of all the events, we reduce the interval to $(\hat{c}, e\,]$. Similarly, if it is to the left, we reduce the interval to $(e, \hat{d}]$. In both cases we compute a new estimate of the location of the event of the first function and repeat. The remaining possibility is that some, but not all, of the event functions have events to the left of e. In this case, we reduce the list of active event functions, and restart with the interval $(\hat{c}, e\,]$ and (possibly) a new first event function.

There are two variations on the basic problem of computing the first event that occur frequently and present difficulties for the software. They involve continuing the integration of (1.1) on to the next event. In the one case, the event functions remain the same, and in the other, they are changed.

A difficulty with the first task is that one event can occur immediately after another. When we compute a position $\bar{t}$, we obtain an interval $(\bar{c}, \bar{d}] \subset (x_i, x_{i+1}]$ such that $\bar{t} \in (\bar{c}, \bar{d}]$ and $(\bar{c}, \bar{d}]$ contains no other events. This is a big help, but there is also the difficulty that events may occur several times in the vicinity of $\bar{t}$ when finite precision arithmetic is used, yet only once in infinite precision arithmetic. Hence, even though $\bar{t}$ is excluded from our interval of search for the position of the next event, a point nearby may be identified incorrectly as that position. To help overcome this problem, we search $(\bar{d}, x_{i+1}\,]$ instead of $(\bar{t}, x_{i+1}\,]$. This search proceeds in two stages. First, we check the remaining part $(\bar{d}, x_{i+1}]$ of the current integration interval $(x_i, x_{i+1}]$ using information computed already. If at least one event occurs in this interval, we compute the position closest to $\bar{d}$ by proceeding as above. Otherwise, we move on to the next integration interval

and proceed as for the case of finding the first position of any event.

For the second problem, the information contained in the computed interval $(\bar{c}, \bar{d}]$ is no longer useful, as after changing event definition there may be an occurrence of one of the newly defined events in $(\bar{t}, \bar{d}]$. Hence we can never avoid possible "numerical roots" without recourse to an error analysis as discussed in Section 2. In our implementation, if we redefine the set of events at a point $\bar{t}\varepsilon(x_i, x_{i+1}]$, then before proceeding we must compute the internal data (for example, Sturm sequence coefficients) on $(x_i, x_{i+1}]$. We may then check for occurrences of the events in $(\bar{t}, x_{i+1}]$. From here on the algorithm proceeds like the one described earlier. If it is possible to work throughout with a set of events that includes all the possible definitions that will arise at any stage in the integration, then from the point of view of, at least sometimes, avoiding rounding effects, this would be the better choice. If this is possible, one should use the algorithm described for the first problem rather than that for the second even though this might involve some postprocessing by the user of the event position information.

4 Combining the Root-Finder with an Integrator

In this section we consider the design of an interface that allows a root- finding code to be grafted onto many popular integrators. A version we plan to disseminate makes minimal assumptions about the integrator. We assume only that it can be used in a mode such that it returns to the calling program after each step from x_i to x_{i+1} with a polynomial of known degree r that is to represent the solution on all of $[x_i, x_{i+1}]$. As noted earlier, anomalous behavior is possible with the interpolants of certain popular codes because they do not connect smoothly at mesh points. This does not interfere with the root-finder in its treatment of the interval $(x_i, x_{i+1}]$, and provided the user appreciates the potential for difficulties arising from the lack of smoothness, there is no reason why he cannot use our root-finder with such a code. The effects of the *lack* of smoothness here are similar, but more marked, than those arising from *loss* of smoothness due to numerical errors in our algorithm. We discuss this further at the end of the section.

The modules we have written for the root-finding task assume that the polynomial is of the form

$$\bar{p}(s) = \sum_{i=0}^{r} a_r s^r, \qquad 0 \le s \le 1. \tag{4.1}$$

Popular integrators represent their interpolating polynomials in many forms, viz., Taylor, Lagrangian, divided difference, Hermite, ... A routine is needed

r	$cond(V)$	r	$cond(V)$
1	4.0E0	8	1.9E6
2	2.4E1	9	1.1E7
3	1.9E2	10	7.2E7
4	1.2E3	11	4.4E8
5	7.7E3	12	2.6E9
6	4.8E4	13	1.6E10
7	3.2E5	14	9.8E10

Table 1: $Cond(V)$ for $r+1$ interpolation nodes

to convert to this standard form. We first consider how to do this when no information about the form is supplied, just the ability to evaluate the interpolant. It is hardly surprising that a better job can be done when more information is supplied. Some examples of this will be taken up.

With just the ability to evaluate the polynomial $p(x)$ of degree r provided by the integrator, we convert it to the form (4.1) by choosing $r+1$ points

$$0 \le s_0 < s_1 < \cdots < s_r \le 1 \tag{4.2}$$

and forming the values

$$\bar{p}(s_i) = p(x_i + s_i(x_{i+1} - x_i)) \tag{4.3}$$

Construction of p in this way involves the solution of a van der Monde system with matrix $V = (s_i^j)$. The computations can be ill-conditioned, especially for the high degrees that sometimes occur with Adams codes. Table 1 gives $Cond(V) = ||V||_\infty ||V^{-1}||_\infty$ for $r = 1, 2, \ldots, 14$ when the s_i are the extrema of the Chebyshev polynomial $T_{r+1}(x)$ shifted to $[0, 1]$. Degrees 1–5 are typical of BDF and Runge-Kutta codes. The higher degrees can occur in Adams codes. The condition numbers are uncomfortably large for the higher degrees. However, they are notoriously pessimistic, and the analysis of [7] shows that we should not expect the size of the condition number to be reflected in the errors in the coefficients a_r in (4.1) as long as we use an appropriate algorithm for solving the van der Monde system.

The general purpose package that we have written uses the nodes specified, except that the user may override the choice. This approach is not all that one might hope for. The Nordsieck form of the interpolating polynomial is used in a number of popular codes. It is just a Taylor series expansion of the polynomial, so a special interface to convert it to the shifted and scaled form (4.1) that we need is much more natural than the general approach. Such a special interface should be less troubled by conditioning. Because the matter is straightforward, we do not go into details.

r	$cond(V)$	r	$cond(V)$
3	5.4E1	9	8.4E6
5	2.9E3	11	4.0E8
7	1.8E5	13	1.7E10

Table 2: $Cond(V)$ for $\bar{r}$ interpolation nodes, both function and derivative

Some routines for evaluating $p(x)$ evaluate $p'(x)$ at the same time. For odd $r = 2\bar{r} - 1$, we might then replace (4.3) above by

$$\bar{p}(\bar{s}_i) = p(x_i + \bar{s}_i(x_{i+1} - x_i)) \tag{4.4}$$

$$\bar{p'}(\bar{s}_i) = p'(x_i + \bar{s}_i(x_{i+1} - x_i)) \tag{4.5}$$

for $i = 1, 2, \ldots, \bar{r}$ and a set of distinct points

$$0 \leq \bar{s}_1 < \bar{s}_2 < \cdots < \bar{s}_{\bar{r}} \leq 1 \tag{4.6}$$

In Table 2 we tabulate $Cond(V)$. The values are a little smaller than those of Table 1, but they increase similarly.

Recently there has been considerable interest in providing Runge-Kutta codes with the kind of polynomial interpolant that we require. The paper [4] provides an entry to the literature. We have proposed [11, 6] an algorithm based on the Dormand-Prince-Shampine (DPS) formulas which are a (4,5) pair of embedded formulas that use local extrapolation. They yield fifth order approximations to the local solution and its derivative at x_i, $\frac{1}{2}(x_{i+1}+x_i)$ and x_{i+1}. A quintic Hermite interpolant then yields a polynomial approximation of order five on $[x_i, x_{i+1}]$ which connects up with approximations on other intervals to yield a globally C^1 piecewise polynomial approximate solution. It is natural here to take the nodes $\{\bar{s}_i\}$ to be $\{0, \frac{1}{2}, 1\}$ in (4.6) and solve directly for $\bar{p}(s)$ using (4.4) and (4.5). We have experimented with a special interface for this and similar cases, as well as the general one presented earlier.

Observe that even when the underlying polynomial $p(x)$ is, in theory, globally C^1, this property will not be shared by the computed piecewise polynomial functions defined by (4.1) for each interval $[x_i, x_{i+1}]$ in turn. This loss of smoothness can arise from inaccurate solution of the van der Monde system on each interval and, where (4.1) is obtained from evaluations of $p(x)$ only, from not imposing the condition $p'(x_i) = y'_i$ directly. It will manifest itself in errors in the locations of the events and, more importantly but far less frequently, in the "loss" of event occurrences at the mesh points as we switch between the polynomials on successive intervals. In general, there is little we can do about this in our codes, but in the driver routine, where we control the move from one interval to the next, we check and report on discontinuities in event location caused

by errors at the mesh points. We do this first by checking for sign changes at the mesh points in the representations of the event functions from the two intervals meeting at this point. If this first approach reveals nothing, we check then that the Sturm sequence counts match for these two representations over a small subinterval $(x_{i+1}, x_{i+1} + \delta]$ of the new interval. Here δ is chosen fairly small, relative to the size of the interval $(x_{i+1}, x_{i+2}]$, as extrapolation with the representation computed on $(x_i, x_{i+1}]$ is likely to be very inaccurate for larger δ, see [6].

5 The Codes

In [2] we present a collection of subroutines implementing the algorithm described in this paper. We refer the reader there for a fuller description of the codes and here we restrict ourselves to some general comments pertinent to the material of this paper. Also we discuss the use of a driver subroutine, ALEVNT, for an important application.

The theoretical discussion above is faithfully reflected in our codes and gives an adequate appreciation of their efficiency. Clearly if we make the usual assumption that evaluating the ODE (1.1) is the expensive part of the integration, then our event location technique constitutes additional overhead of comparatively low cost. In particular, it is normally the case that no evaluations of the ODE are required in our algorithm over and above those required for the integration alone. Here we are assuming, as is common in most modern codes, that calculating the interpolant associated with the ODE solver does not involve further ODE evaluations. Even if it did, any extra evaluations would be made *once per step* rather than once per event function evaluation. After an interval containing an event is located, our algorithm is nearly as efficient as the root finder on which it is based when used in the standard way to calculate a root of a single equation.

In addition to the computed event locations our codes return "condition numbers" based on an extension of (2.1) to take account of high order zeros or turning points. For an event function $g = y_j(x) - \alpha$ where the event $\bar{\tau}$ is an occurence of order m, that is, $g^{(r)}(\bar{\tau}) = 0, r = 0, 1, \ldots, m-1$ and $g^{(m)}(\bar{\tau}) \neq 0$, (2.1) gives

$$|\tau - \bar{\tau}| \simeq \{m!(y_j(\tau) - p_j(\tau))/p_j^{(m)}(\bar{\tau})\}^{1/m}. \tag{5.1}$$

Similarly if $g = y_j'(x)$ and $\bar{\tau}$ is an occurrence of order m of this event, then

$$|\tau - \bar{\tau}| \simeq \{m!(y_j'(\tau) - p_j'(\tau))/p_j^{(m+1)}(\bar{\tau})\}^{1/m}. \tag{5.2}$$

On locating an event our codes return both m and the "condition number" $\{m!/p_j^{(m)}(\tau)\}^{1/m}$ or $\{m!/p_j^{(m+1)}(\bar{\tau})\}^{1/m}$ as appropriate. Note that due to rounding and integration error effects, the codes may return too small a value for m. In these cases, in compensation, there will be closely clustered roots and $p_j^{(m)}(\bar{\tau})$

in (5.1) or $p_j^{(m+1)}(\bar{\tau})$ in (5.2) will thus be small, hence the condition number will be large.

Consider the problem of tabulating the independent variable x in terms of equispaced values of a dependent variable y. That is, we wish to solve (1.1) locating the first points x_i such that $y_i = y_0 + i\Delta, i = 0, 1, \ldots$, for a given value y_0 and a given increment Δ. We can pose this problem in two ways. In the first approach we define the single event function to be $g = y - y_0$ and proceed to compute the position x_0 of this event. We then redefine the event to be $g = y - y_1$, compute x_1, and so on. Since it is necessary to redefine only the event function and not the differential equation, we can use subroutine ALEVNT as in Figure 1. The second approach is to define a set of event functions simultaneously, namely $g_j = y - y_j,\ j = 0, 1, \ldots, m$, for all the tabulation points of interest. Then as the events are located, they must be processed to maintain the order of tabulation. The subroutine ALEVNT can be used for this purpose too. It is designed to drive an integrator of the user's choice and to determine the locations of all the events. There is an important distinction between the approaches. If, for example, we start with $y_0 = 1$, and we wish to know where $y(x)$ has values 2, 3, 4, etc. , it is quite possible that in the first approach we do not know which definition to use for the next event function. This happens when we find, for example, where $y(x) = 3$ and wish to continue. Should we look for $y(x) = 4$ as the next event, or $y(x) = 3$ again? The first approach is satisfactory only when we have *a priori* information about the behavior of the solution which we expect to be reflected in the computed solution. Even when we believe we have this information, a useful check on the problem formulation and its coding is to use the second approach. Possibly it should be modified to include only those event functions which could occur starting from the current position (that is three event functions in almost all cases).

As an example, consider the problem of tabulating x at equispaced steps in the solution y of the artificially constructed equation

$$y' = -y^2 + x^6 - 2x^5 + x^4 + 3x^2 - 2x,\ y(-1) = -2.$$

This equation has solution $y = -x^2(1 - x)$ and we tabulate the values of x at which $y = -1, 0, 1, 2$ using the NAG code D02NBF as the integrator with relative local error tolerance 10^{-5}. (We would expect these results to be reproduced approximately by any standard integrator.) We can define just one event function at a time in ALEVNT and after each event has been located redefine the event function. Alternatively, we can use the *four* event functions corresponding to the *four* tabulation points simultaneously. In Table 3 we present the tabulated values for the latter approach and an error estimate based on substituting the local error tolerance for the expression $y_j(\tau) - p_j(\tau)$ in (5.1). Though the value substituted should be the global error in y_j, the local error tolerance is used because it is the only quantity available which is at all related to the error in y_j. Though not well established, the error estimate is clearly a

y	Tabulation Point	Multiplicity	Error Estimate
-1	-0.07549	1	0.808E-4
0	-0.00022	1	0.135
0	0.00023	1	0.135
0	0.99999	1	0.555E-4
1	1.46557	1	0.158E-4
2	1.69562	1	0.106E-4

Table 3: Tabulation points for equispaced values of y.

reasonable approximation to the true error at $x = 1$, but an overestimate for the error of the approximations to the double zero at $x = 0$. We have computed these same events using different error tolerances in the call to D02NBF. For all sufficiently stringent tolerances we obtain the underlined event locations as given in Table 3. In some cases we detect only one event $y = 0$ and we never detect a multiple root. Even though y has a root of multiplicity two at $y = 0$, the numerical approximation to y has either no roots or a close pair at that value. The error estimates associated with the close pair are consistent with the error tolerances in the integration and always indicate ill–conditioned roots. The results illustrate the value of using all four event functions simultaneously. They reveal that y is not monotonic. The results for the first approach above, using just one event function, are exactly the underlined values in Table 3.

Clearly results obtained by redefining the event functions and restarting the integration are not, in general, likely to be as accurate as those calculated without restarting the integration. In the case of restarting, the errors made in calculating the early event locations will propagate into the later integrations, hence into the accuracy of the later event locations; in the case of not restarting they have no effect.

6 Conclusions

We have outlined an approach to finding all the event locations of certain special event functions associated with the solution of an ODE. In addition to emphasizing the difficulty of the problem, we have shown how we construct our event locating code and we have outlined how we graft such a code onto a standard integrator with an interpolation feature. This grafting process requires of the integrator only that it have a step oriented mode, that we can determine the degree of the associated interpolating polynomial on each step, and that we can evaluate the interpolating polynomial anywhere in the span of the step.

```
/* A pseudo-code redefining event functions as locations are detected
      { Declarations }
      { User initializes integrator, sets t := x1 := a and tend }
      { User defines first event: set neqg = 1, turn(1) = .false. and
        comps(1) and alpha(1) appropriately }
      { irevcm is a reverse communication integer variable
        set by ALEVNT before exit to determine the next
        action required of its user}
      irevcm := 0
      call ALEVNT(neqg, comps, alpha, turn, x1, x2, nord, mxord,
                  neqf, xval, yvals, lbound, xevent, rbound, mltplc,
                  cndnum, typtrn,... workspace ..., irevcm, ier)
      { variables x1, ..., yvals are input values to ALEVNT
        (which are output from the integrator) and variables
        lbound, ..., typtrn are the output variables from
        ALEVNT}
      if (irevcm = 0)
          { User checks ier for nature of error }
      elseif (irevcm = 1)
          { User calls integrator to take first integration
            step (x1, x2) }
          goto call of ALEVNT
      elseif (irevcm = 2 and t < tend)
          { User calls integrator to take integration step }
          goto call of ALEVNT
      elseif (irevcm = 3)
          { User evaluates interpolant at xval in yvals }
          goto call of ALEVNT
      elseif (irevcm = 4)
          print xevent
          irevcm := 5
          { User resets comps(1) and alpha(1) to define next event }
          goto call of ALEVNT
      endif
/*End
```

Figure 1. Using ALEVNT when redefining the event function

References

[1] M. Berzins, A C^1 Interpolant for Codes Based on Backward Difference Formulae. *Appl. Numer. Math.*, **2** (1986) 109–118

[2] R.W. Brankin, I. Gladwell and L.F. Shampine, Codes for Reliable Solution of Special Event Location Problems for ODEs, Num. Anal. Rept. 139, Dept. of Math., University of Manchester, 1987.

[3] R.P. Brent, *Algorithms for Minimisation Without Derivatives*, Prentice-Hall, Englewood Cliffs, NJ, 1973.

[4] J.R. Dormand and P.J. Prince, Runge-Kutta Triples, *Comp. and Maths. with Appls.*, **12** (1986) 1007–1017.

[5] C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1971.

[6] I. Gladwell, L.F. Shampine, L.S. Baca and R.W. Brankin, Practical Aspects of Interpolation in Runge-Kutta Codes, *SIAM J. Sci. Stat. Comput.*, **8** (1987) 322–341.

[7] N.J. Higham, Error Analysis of the Bjorck-Pereyra Algorithms for Solving van der Monde Systems, *Numer. Math.*, **50** (1987) 613–632.

[8] M.K. Horn, Fourth and Fifth-Order Scaled Runge-Kutta Algorithms For Treating Dense Output, *SIAM J. Numer. Anal.*, **20** (1983) 558–568.

[9] A Ralston and P. Rabinowitz. *A First Course in Numerical Analysis* (2nd edition). McGraw Hill, New York, 1978.

[10] L.F. Shampine and M.K. Gordon, *Computer Solution of Ordinary Differential Equations: the Initial Value Problem*, W.H. Freeman, San Francisco, 1975.

[11] L.F. Shampine, Some Practical Runge-Kutta Formulas, *Math. Comput.*, **46** (1986) 135–150.

[12] H.A. Watts, RDEAM—An Adams ODE Code with Root Solving Capability, Rept. SAND85–1595, Sandia National Laboratories, Albuquerque, NM87185, 1985.

[13] H.A. Watts, Backward Differentiation Formulae Revisited: Improvements in DEBDF and a New Root Solving Code RDEBD, Rept. SAND85–2676, Sandia National Laboratories, Albuquerque, NM 87185, 1986.

[14] H.A. Watts and L.F. Shampine, Smoother Interpolants for Adams Codes, *SIAM J. Sci. Stat. Comput.*, **7** (1986) 334–345.